Paper Number: P1830R1
Title: `std::dependent_false`
Authors: Ruslan Arutyunyan <Ruslan.Arutyunyan@intel.com>

With Input from: Billy O'Neal <bion@microsoft.com>,
CJ Johnson <johnsoncj@google.com>
Audience: LEWG-I (Library Evolution Working Group-Incubator)
Date: 2019-10-03

I. Introduction

We need to introduce a generic solution to create the dependent scope for `static_assert(false)` expression where it is semantically necessary.

II. Motivation and Scope

In several scenarios `static_assert(false)` expression is a useful construction. That happens when better diagnostics should be provided to the user. However, if implementer just writes `false` as the first argument of the `static_assert` the program has never been compiled successfully.

Consider the following examples where such semantics can be useful:

Suppose the user have to implement the function with the signature:

```
template <typename T>
int my_func(const T&)
```

and it is necessary to implement it in accordance with the following requirements:

- If T is integral type, returns 1
- Otherwise if T is convertible to `std::string`, returns 2
- Otherwise the program is ill-formed.

Possible implementation might be:

```
template <typename T>
int my_func(const T&)
{
    if constexpr(std::is_integral_v<T>)
    {
        return 1;
    }
    else if constexpr (std::is_convertible_v<std::string, T>)
    {
        return 2;
    }
    else
    {
        // Always Compile-time error
        static_assert(false, "T is not integral and is not
                              convertible to std::string");

    }
```

```
}
```
But as mentioned above this code cannot be compiled successfully due to `static_assert(false)` expression.

Another example where `static_assert(false)` might be useful is the class template for which primary template is not defined. Instead, user should always pass correct template arguments that one of specializations has been chosen.

Consider the following code snippet:

```
// Primary template
template <typename T, typename U>
struct my_struct;

// Partial specialization
template <typename T, typename Alloc>
struct my_struct<int, std::vector<T, Alloc>>
{

};

// User code
int main()
{
    my_struct<int, int> s;
}
```

Examples of compiler messages are:

- Clang: `error: implicit instantiation of undefined template 'my_struct<int, int>'`
- GCC: `error: aggregate 'my_struct<int, int> s' has incomplete type and cannot be defined`
- Intel Compiler: `error: incomplete type is not allowed my_struct<int, int> s;`

Implementer might want to provide better diagnostics to the user. The possible approach might be:

```
template <typename T, typename U>
struct my_struct
{
    // Always Compile-time error
    static_assert(false, "Type T and Type U cannot be used in such
                          combination. See the documentation");
};
```
Unfortunately, the static assertion in the code above is always failed despite if primary template has been chosen or not.

III. Problem statement

To overcome the mentioned issue the implementer should write some implementation to create dependent scope for the `static_assert(false)` expression.

Example:

```
template <typename T>
constexpr bool always_false()
{
    return false;
}

template <typename T, typename U>
struct my_struct
{
    // static_assert fails only if primary template is chosen
    static_assert(always_false<T>());
};
```

Many template libraries implement the approach above in their manner. It's better to have one standard solution instead of having a lot of workarounds everywhere implemented differently.

IV. Proposal

The issue may be addressed by introducing the generic solution for such problem. The proposed solution based on the previous discussion and analyzed use-cases

dependent_bool_value variable template

```
template <bool value, typename... Args>
inline constexpr bool dependent_bool_value = value;
```

Since the vast majority of use-cases is the necessity to instantiate exactly the dependent_false the following helper variable template is proposed:

```
template <typename... Args>
inline constexpr bool dependent_false = dependent_bool_value<false,
                                                              Args...>;
```

In that case the `static_assert` would look like either:

```
template <typename T, typename U>
struct my_struct
{
    static_assert(dependent_bool_value<false, T>);
};
```

or even simpler with the helper:

```
template <typename T, typename U>
struct my_struct
{
    static_assert(dependent_false<false, T>);
};
```

A dependent static assertion is created with help of `dependent_bool_value` variable template. It would be evaluated only if primary template is chosen.

The proposed API uses variadic templates for dependent context creation. The example below shows why it is convenient. Let's use `dependent_false` helper API as the main use-case:

Suppose that we have `my_struct` declaration as follows:

```
template <typename... Args>
struct my_struct;
```

and the API for dependent scope is

```
template <typename T>
inline constexpr bool dependent_false = dependent_bool_value<false, T>;
```

In that case `my_struct` definition is

```
template <typename... Args>
struct my_struct
{
    static_assert(dependent_false<std::void_t<Args...>>);
};
```

As you can see user needs `std::void_t` (or something else) to transform variadic templates to the one type parameter.

With the proposed API the variadic templates can be directly passed to the `dependent_false` helper:

```
template <typename... Args>
struct my_struct
{
    static_assert(dependent_false<Args...>);
};
```

Since variadic templates can be empty, `Args`... may be missed by mistake but such kind of error would be easily caught at compile-time.

V. Additional notes

a) Non-type template parameters

Current API of `dependent_bool_constant` and `dependent_false` covers the dependent scope with non-type template parameters. See the example below:

```
template <std::size_t N>
void function()
{
    if constexpr (N == 0)
    {
        static_assert(dependent_false<decltype(N)>, "N shall be > 0");
    }
};
```

b) template template parameters

The support of template template parameters looks impossible. Consider the following example.

```
template <typename T, std::size_t, template <typename, int>
              typename Clazz>
class Strange {};

template <template <typename, std::size_t, template <typename, int>
                        typename> typename Clazz>
void func()
{
     // Cannot use dependent_false
}

// call foo with Strange
foo<Strange>();
```

It's hard to impossible to create such API of `dependent_false` that works with any template template parameters count and combination.

Legal Disclaimer & Optimization Notice

Optimization Notice